Composition of Partially Observable Services Exporting their Behaviour

Giuseppe De Giacomo, Riccardo De Masellis, Fabio Patrizi
Dipartimento di Informatica e Sistemistica
SAPIENZA - Università di Roma
Via Ariosto 25 - 00185 Roma, Italy
{degiacomo,demasellis,patrizi}@dis.uniroma1.it

Abstract

In this paper we look at the problem of composing services that export their behavior in terms of a transition system, characterizing the choices of actions given to a client at each point in time. The composition consists of synthesizing an orchestrator that coordinates the available services so as to mimic the desired target service asked by the client. Specifically, in this paper we study the "conformant form" of the problem, where available services are partially controllable and partially observable, and hence, the orchestrator has to make its decisions exploiting the observations made so far only. We give a sound and complete procedure to synthesize the orchestrator in such case, and characterize the computational complexity of the problem. The procedure is based on working with belief (or knowledge) states, a standard technique to tackle conformant planning. Moreover we show that, although in general unavoidable, the powerset construction at the base of the belief state approach can be delegated to the symbolic manipulations of the game-structure model checking tool (TLV), which can be used to efficiently implement the orchestrator synthesis procedure.

Introduction

Planning, in its classical formulation, is probably the most long standing and successful example of automated program synthesis in CS and AI. However, several other forms of automated program synthesis are increasingly attracting the scientific community. These include automated synthesis of reactive systems from logical specifications, e.g., expressed in LTL, which is quickly becoming one of the main new frontiers for model checking (Vardi 2008). Also, logics such as ATL (Alur, Henzinger, and Kupferman 2002), where checking a property amounts to check for the possibility of synthesizing a strategy (which is a sort of program/plan), are considered an important foundation of the multi-agent system formal specification (van der Hoek, Roberts, and Wooldridge 2007). Moreover, several variants of automated service composition synthesis are being investigated in the CS and AI literature, some of which considering services as atomic and using off-the-shelf Planning technology, e.g., (Klusch and Gerber 2006; Hatzi et al. 2008), but also considering services characterized by a complex behaviours modelling agents, devices, components and modules, e.g., (McIlraith and Son 2002; Berardi et al. 2005; Pistore, Traverso, and Bertoli 2005; Hull 2005; Sardiña, De Giacomo, and Patrizi 2008; Lustig and Vardi 2009). While Planning results cannot be directly applied in the latter cases, the vast literature on Planning can shed some light on issues typically faced in this form of automated program synthesis as well, and suggest applicable techniques and solutions (as in the case of the present work).

In this paper, we look at the problem of composing services that export their behavior in terms of a (finite-state) transition system, characterizing the choices of actions given to the client at each point in time. Services can be thought of as reactive components that, step by step, can be asked for executing one action among those available in their current state and that, as a result of action execution, change their current state, thus making available a new set of executable actions. Notice that, services are not necessarily meant to terminate, since may describe (interactive) behaviors that are supposed to run forever. So, while transition systems are finite, the behaviours they describe is infinite, i.e. they contain loops. The composition task we are interested in consists of synthesizing an orchestrator that coordinates the available services and repurposes them by extracting suitable fragments, so as to mimic a desired target service asked by a client, as long as asked by the client (possibly forever) (De Giacomo and Sardiña 2007; Sardiña, De Giacomo, and Patrizi 2008).

Interesting connections exist between such a form of composition and Planning, in particular wrt the basic notions of operators, goal, and plan. *Operators*, instead of being atomic actions as typical in Planning, are full fledged *services* represented as *transition systems*. The *goal*, instead of being a state of affair to reach, is, again, a full fledged service, called *target service*, describing the behavior that the client desires to interact with. A *plan*, instead of being a controlled *sequential composition* of available operators, is a controlled *concurrent composition* of available services, called *orchestrator*. Solving the *planning problem* amounts to synthesize a orchestrator which is able to maintain over time the client's ability of choosing the desired action, instead of synthesize a sequence of actions able to reach a desired state of affair as typical in Planning.

Planning comes in several fundamental forms, namely, "classical", "conditional", and "conformant" (Ghallab, Nau, and Traverso 2004). We start our work by observing that, such distinctions can be applied also to our problem. In

the *classical form* –the simplest case– available service is deterministic and hence fully controllable by the orchestrator. This form was first studied in the context of service composition (Berardi et al. 2005). In the *conditional form*, available services are nondeterministic and hence only partially controllable. As a consequence, the orchestrator has to sense/observe the resulting services' states before deciding what to do next. This form of composition has been studied mainly in the AI literature, in the context of agents' and devices' composition (De Giacomo and Sardiña 2007; Sardiña, De Giacomo, and Patrizi 2008; Stroeder and Pagnucco 2009). Finally, in the *conformant form*, available services are not only partially controllable, but also partially observable, hence the orchestrator has to make its decisions based on the observations made so far only.

In this paper we give two main contribution. The first contribution is a sound and complete procedure to synthesize an orchestrator in the conformant case, and characterize the computational complexity of the problem. Our procedure is based on working with belief (or knowledge) states, a standard technique to tackle conformant planning (Goldman and Boddy 1996), which however is typically sound but incomplete when dealing with infinite/unbounded behaviors, as in our case (Sardiña, De Giacomo, and Patrizi 2008) (cf. Conclusions). Here, instead, we show that the technique is indeed complete for the problem at hand and in fact optimal wrt the computational complexity. The second contribution has a more practical flavor, we show that, although in general unavoidable, the powerset construction at the base of the belief state approach can, in our case, be delegated to the symbolic manipulations of the game-structure model checking tool (TLV) (Piterman, Pnueli, and Sa'ar 2006), which is used for efficiently implement the orchestrator synthesis.

Preliminaries

Here we briefly summarize the framework proposed in (De Giacomo and Sardiña 2007; Sardiña, De Giacomo, and Patrizi 2008).

Services. A service, at each step, offers to its clients a choice of actions, based on its state; the client chooses one of them, and the service executes it, moving to a new service state. Formally, a *service* is a tuple $\mathcal{S} = \langle \mathcal{A}, S, s_0, S^f, \varrho \rangle$, where:

- $\mathcal{A}$ is a set of actions;
- S is the finite set of service's states;
- $s_0 \in S$ is the initial state;
- $S^f \subseteq S$ is the set of *final* states, i.e., those where the execution can be legally stopped (if desired by the client);
- $\varrho \subseteq S \times \mathcal{A} \times S$ is the service's transition relation.

When $\langle s, a, s' \rangle \in \varrho$, we say that *transition* $s \xrightarrow{a} s'$ *is in* $\mathcal{S}$. Given a state $s \in S$, if there exists a transition $s \xrightarrow{a} s'$ in $\mathcal{S}$ (for some s') then action a is said to be *executable* in s. A transition $s \xrightarrow{a} s'$ in $\mathcal{S}$ denotes that s' is a possible successor state of s, when action a is executed in s. If an action a is not executable in s, we write $s \xrightarrow{a} /$. Available services are, in general, *nondeterministic*, that is, they allow many transitions to take place under execution of a same action. So, when choosing the action to execute next, the client of the service cannot be certain of which choices will be available later on, this depending on which transition actually takes place. In other words, nondeterministic behaviors are only *partially controllable*.

We say that a service $\mathcal{S}$ is *deterministic* iff there not exist, in $\mathcal{S}$, two distinct transitions $s \xrightarrow{a} s'$ and $s \xrightarrow{a} s''$ such that $s' \neq s''$. Notice that given a deterministic service's state and an executable action in that state, the *unique* next service state is always known. That is, deterministic services are indeed *fully controllable* by selecting actions.

Obviously, deterministic services are *client-friendly*, being fully controllable by client's actions, while this is not the case for nondeterministic ones. The need for dealing with nondeterministic services comes from abstraction: the actual state resulting from a transition may depend on details that, for various reasons, are kept hidden to the client, which, as a result, sees the service as nondeterministic.

Composition. The composition problem is the following: given a *target service*, which is a *deterministic* service required by the client, and given a set $\mathcal{S}_\mathcal{C} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$, called *community*, of possibly nondeterministic available services (which share the same actions as the target), realize the target services, by properly *delegating the action requested to the target to the available services*.

Figure 1 shows an example of a target service and a community formed by two available services (dotted lines will become relevant later).

The device that is in charge of the delegation is the *orchestrator*, which is a component that able to activate, stop and resume each of the available services, and select one to perform an executable action. At every point in time, the orchestrator intercepts the action requested by the client (according to the target service), selects one of the available services, and delegates to it the action requested.

In order to formally define orchestrator and composition, we need to introduce the notion of traces and histories. Given a service $\mathcal{S} = \langle \mathcal{A}, S, s_0, S^f, \varrho \rangle$, a *trace* for $\mathcal{S}$ is a possibly infinite sequence, alternating configurations and actions, of the form $\langle s^0 \rangle \xrightarrow{a^1} \langle s^1 \rangle \xrightarrow{a^2} \cdots$, such that *(i)* $\langle s^0 \rangle = \langle s_0 \rangle$, and *(ii)* for all $j > 0$, we have that $s^j \xrightarrow{a^{j+1}} s^{j+1}$ is in $\mathcal{S}$. Similarly, let $\mathcal{S}_\mathcal{C} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ be a community, where $\mathcal{S}_i = \langle \mathcal{A}, S_i, s_{i,0}, S_i^f, \varrho_i \rangle$ $(i = 1, \ldots, n)$. A *community trace* for $\mathcal{S}_\mathcal{C}$ is a possibly infinite sequence of the form $\langle s_1^0, \ldots, s_n^0 \rangle \xrightarrow{a^1, v^1} \langle s_1^1, \ldots, s_n^1 \rangle \xrightarrow{a^2, v^2} \cdots$, such that *(i)* $\langle s_1^0, \ldots, s_n^0 \rangle = \langle s_{1,0}, \ldots, s_{n,0} \rangle$, and *(ii)* for all $j > 0$, if $\langle s_1^j, \ldots, s_n^j \rangle \xrightarrow{a^{j+1}, v^{j+1}} \langle s_1^{j+1}, \ldots, s_n^{j+1} \rangle$, then $s_{v^{j+1}}^j \xrightarrow{a^{j+1}} s_{v^{j+1}}^{j+1}$ in $\mathcal{S}_{v^{j+1}}$ with $s_i^{j+1} = s_i^j$ for $i \neq v^{j+1}$. We call (community) *history* every finite prefix of a (community) trace ending with a configuration. Given a history h, we denote by $last(h)$ the last configuration, and by $length(h)$ the number of alternations between configurations and actions in h. Notice that the history of length 0 is simply the initial configuration of a trace (which is the same for every trace).

Let $\mathcal{S}_\mathcal{C} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ be a community and $\mathcal{H}$ be the set of its community service histories. Formally an *orchestrator* for a community $\mathcal{S}_\mathcal{C}$ is a function $P : \mathcal{H} \times \mathcal{A} \to \{1, \ldots, n, u\}$ that, given a history $h \in \mathcal{H}$ and an action $a \in \mathcal{A}$, selects an available service, i.e., returns its index, to which to delegate a. Special value u is introduced for technical convenience, to make function P total.

Let $\mathcal{S}_\mathcal{C} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ be a community and $\mathcal{S}_t$ a target service, where, $\mathcal{S}_i = \langle \mathcal{A}, S_i, s_{i,0}, S_i^f, \varrho_i \rangle$ $(i = t, 1 \ldots, n)$. Let $P : \mathcal{H} \times \mathcal{A} \to \{1, \ldots, n, u\}$ be an orchestrator for $\mathcal{S}_\mathcal{C}$. Given a trace $\tau = \langle s^0 \rangle \xrightarrow{a^1} \langle s^1 \rangle \xrightarrow{a^2} \cdots$ of $\mathcal{S}_t$, we say that the *orchestrator P realizes the trace τ* if

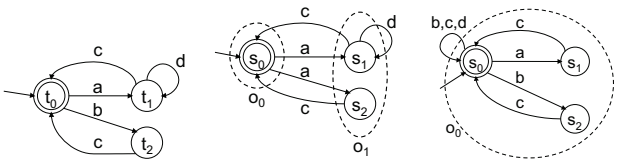
- for all community service histories $h \in \mathcal{H}_\tau$, $P(h, a^{length(h)+1}) \neq u$, where $\mathcal{H}_\tau = \bigcup_\ell \mathcal{H}_\tau^\ell$ is a set of community service histories, inductively defined as follows: *(i)* $\mathcal{H}_\tau^0 = \{\langle s_{1,0}, \ldots, s_{n,0} \rangle\}$; *(ii)* $\mathcal{H}_\tau^{j+1}$ is the set of community histories of length $j+1$ having the form $h' = h \xrightarrow{a^{j+1}, v^{j+1}} \langle s_1^{j+1}, \ldots, s_n^{j+1} \rangle$ such that:
  - $h \in \mathcal{H}_\tau^j$, with $last(h) = \langle s_1^j, \ldots, s_n^j \rangle$;
  - a^{j+1} is the action in history of length $j+1$ obtained from τ.
  - $P(h, a^{j+1}) = v$, that is, the orchestrator states that action a^{j+1} in the trace τ after community history h should be executed by available service $\mathcal{S}_v$;
  - $s_v^j \xrightarrow{a^{j+1}} s_v^{j+1}$ in $\mathcal{S}_v$ that is, the available service $\mathcal{S}_v$ can evolve according to the history h'.
  - $s_i^{j+1} = s_i^j$ for each $i \neq v$
- if a configuration $\langle s_t^\ell \rangle$ of τ is such that $s_t^\ell \in \mathcal{S}_t^f$, then every configuration $\langle s_1^\ell, \ldots, s_n^\ell \rangle = last(h)$, with $h \in \mathcal{H}_\tau^\ell$, is such that $s_i^\ell \in \mathcal{S}_i^f$, for $i = 1, \ldots, n$.

We observe that each $\mathcal{H}_\tau^j$ contains all and only those community service histories of length j that the orchestrator P may potentially generate when instructing available services so as to execute/realize a prefix of trace τ of length j. The set $\mathcal{H}_\tau$ is the union of all such $\mathcal{H}_\tau^j$. Also we observe that, in order for an orchestrator P to realize τ, it must be able to instruct one of the available services to execute each action in τ whatsoever is the corresponding history of the enacted community service. Finally we note that, in the definition above target service states are never mentioned. This is unsurprising being the target service deterministic, and, consequently, its states fully determined by the sequence of actions performed in the trace.

An orchestrator P for $\mathcal{S}_\mathcal{C}$ is a *composition* for the target service $\mathcal{S}_t$ iff it *realizes all traces* of $\mathcal{S}_t$. For simplicity, and w.l.o.g., we further require here that $P(h, a) = u$ for all histories $h \notin \mathcal{H}_\tau$ for any trace τ of $\mathcal{S}_t$, and for all actions a not executable in $\mathcal{S}_t$ after the sequence of actions in h.

Intuitively, the orchestrator realizes a target service if for all target service traces, at every step, it returns the index of an available service that can actually perform the requested action. Observe that since available services are nondeterministic, the orchestrator must be always able to execute the next action, no matter how the activated service happen to evolve after each step.

Checking the existence of a composition is an EXPTIME-complete problem, where the exponentially depends on the number of available services in the community (not on the number of their states) (Sardiña, De Giacomo, and Patrizi 2008). Several techniques have been proposed to actually compute the composition, i.e., to synthesize an orchestrator that is a composition, originally based on reduction to PDL satisfiability (Berardi et al. 2005), more recently based on a variant of the classical notion of simulation (Sardiña, De Giacomo, and Patrizi 2008) and on direct algorithms (Stroeder and Pagnucco 2009). Here (see Section Implementation) we use a technique based on the reduction to symbolic model checking of a game structure (Patrizi 2009), which is strongly related to the solutions based on simulation mentioned above.



(a) $\mathcal{S}_t$ (b) $\mathcal{P}_1$ (c) $\mathcal{P}_2$

Figure 1: POS Community.

Partial Observability

We now drop the assumption of *full observability* of the states of the available services in the community and assume that the orchestrator cannot access the available service states but can only access what is explicitly observable, thus it has to deal with *partial observability* of the available service state. A partially observable service, still represented as a transition system, with the addition of an observability function σ, defined over the set of a service's states. Notice that, two states may have the same value of observation, and if they do, they are indistinguishable through direct observation (though, they may still be distinguishable by the history if observation so far).

Formally, a *partially observable service* (POS) is a tuple $\mathcal{P} = \langle \mathcal{A}, \mathcal{O}, S, s_0, S^f, \varrho, \sigma \rangle$ where:

- $\mathcal{A}$, S, s_0, ϱ, are as above;
- $\mathcal{O}$ is the set of observation values;
- $\sigma : S \to \mathcal{O}$ is a function that returns the observation of a state. If this function is injective, we say that the service is fully observable.

In Figure 1 a POS community is shown. Dotted lines group states with same observation.

POS composition. An *observed trace* for a POS community $\mathcal{P}_\mathcal{C}$ is the result obtained, after applying the σ-observation function at every community state, of a standard trace $\tau_{st,\mathcal{P}_\mathcal{C}} = \langle s_1^0, \ldots, s_n^0 \rangle \xrightarrow{a^1, v^1} \langle s_1^1, \ldots, s_n^1 \rangle \xrightarrow{a^2, v^2} \ldots$, so a possibly infinite sequence of the form $\tau_{obs,\mathcal{P}_\mathcal{C}} = \langle o_1^0, \ldots, o_n^0 \rangle \xrightarrow{a^1, v^1} \langle o_1^1, \ldots, o_n^1 \rangle \xrightarrow{a^2, v^2} \ldots$, where $\langle o_1^i, \ldots, o_n^i \rangle = \langle \sigma(s_1^i), \ldots, \sigma(s_n^i) \rangle$. An *observed history* a finite prefix of an observed trace.

Given a standard history h_{st}, we can obtain the associated observed history by simply applying the σ function to each state of the community. We denote the resulting observable history by $obs(h_{st})$. Vice versa, given an observed history h_{obs} representing the evolution of a POS community $\mathcal{P}_\mathcal{C}$, we denote the set $std(h_{obs})$ of all standard histories that represents all possible evolutions of $\mathcal{P}_\mathcal{C}$ with the same observed history h_{obs}. Such a set is defined as $std(h_{obs}) = \{h_{st} \in \mathcal{H}_{st} \mid obs(h_{st}) = h_{obs}\}$.

Let $\mathcal{P}_\mathcal{C}$ be a POS community, $\mathcal{H}_{obs}$ the set of its observed histories, and $\mathcal{A}$ the shared operation alphabet. An *partially observable services orchestrator* (or *POS orchestrator* for short) for $\mathcal{P}_\mathcal{C}$ is a function $P_{obs} : \mathcal{H}_{obs} \times \mathcal{A} \to \{1, \ldots, n, u\}$.

Given a POS orchestrator P_{obs} we can define a standard orchestrator $\bar{P}_{st}$ associated to P_{obs}, called *generated standard orchestrator*, as $\bar{P}_{st}(h_{st}, a) \doteq P_{obs}(obs(h_{st}), a)$.

A POS orchestrator P_{obs} of $\mathcal{P}_\mathcal{C}$ is an *partially observable services composition* (or *POS composition*) for the target service $\mathcal{S}_t$ iff its generated standard orchestrator $\bar{P}_{st}$ is a composition for the target service $\mathcal{S}_t$.

Notice that *we have defined POS compositions exploiting directly the definition of the standard one, through the use of generated standard orchestrator*. Obviously the generated standard orchestrator $\bar{P}_{st}$, even if it is a standard orchestrator, behaves in a specific way, i.e., as a POS orchestrator, returning the same index for all standard histories that have the same observation.

Belief State Composition

Next, we show how to synthesize a composition in the partially observable case. We do so by transforming a community of partially observable services into a new fully observable one. We show that this transformation preserves the soundness and the completeness of the search for a composition. Using such transformation we can directly exploit the compositions techniques that assume full observability, mentioned in Section Preliminaries.

KS. The idea that's behind the transformation of a POS into a fully observable service is to keep track of the *belief states* (cf. (Goldman and Boddy 1996)) the orchestrator goes through while executing a POS. The resulting (belief state) service or *KS*, is obtained by applying two main concepts: *(i)* every time we don't know in which state the original service is, we model our incomplete knowledge by creating a new belief state containing all states in which the service could be, and *(ii)* all information we can infer about the new belief state is only what can be confirmed in *all* states belonging to the belief state. Obviously, belief states are composed of states sharing the same observation.

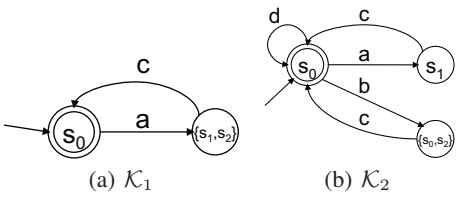
In order to define formally define KS, we need the following preliminary definitions. Let $\mathcal{P} = \langle \mathcal{A}, \mathcal{O}, S, s_0, S^f, \varrho, \sigma \rangle$ be a POS, then $\forall k \in 2^S, \forall a \in \mathcal{A}$ we define:

$$img(k,a) = \begin{cases} \emptyset, \text{ if } \exists s \in k \text{ such that } s \overset{a}{\not\to} \\ \{s' \in S \mid \exists s \in k \wedge s \xrightarrow{a} s'\}, \text{ otherwise} \end{cases}$$

$$obs(k) = \{o \mid \exists s \in k \wedge \sigma(s) = o\}$$

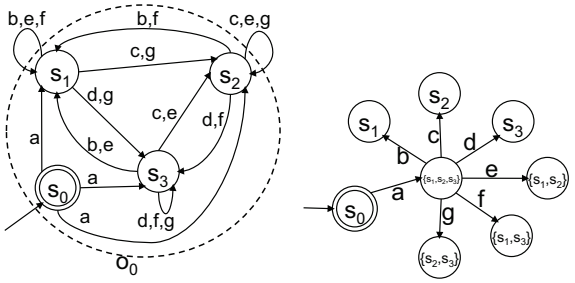
Intuitively, $img(k, a)$ (*image of k wrt a*) is the set of states that can be reached by performing the action a starting from one of the states in k. This set is empty if there is at least one state in which a cannot be performed. Set $obs(k)$ (*observations of k*) is the set resulting from the application of function σ to all states in k.

Given a POS $\mathcal{P} = \langle \mathcal{A}, \mathcal{O}, S, s_0, S^f, \varrho, \sigma \rangle$ we define the associated KS as the fully observable service $\mathcal{K} = \langle \mathcal{A}, K, k_0, K^f, \varrho_k \rangle$ where:



(a) $\mathcal{K}_1$ (b) $\mathcal{K}_2$

Figure 2: KS Community associated with that of Figure 1.



(a) Original POS (b) Transformed exponential KTS

Figure 3: Example of Worst Case Transformation.

- the set of states $K \subseteq 2^S$ is built inductively as follows:
  - $\{s_0\} \in K$;
  - if $k \in K$ and $k_o = \{s \in img(k, a) \mid \sigma(s) = o\}$ then $k_o \in K$;
- the transition relation $\varrho_k \subseteq K \times \mathcal{A} \times K$ is built as follows: $k \xrightarrow{a} k'_o$ if and only if: $k'_o = \{s \in img(k, a) \mid \sigma(s) = o\}$ for some $o \in obs(img(k, a))$;
- $k_0 = \{s_0\}$;
- $K^f = \{k \in K \mid \forall s_i \in k$ we have $s_i \in S^f\}$.

For technical convenience we also define an observation function $\sigma_k : K \to \mathcal{O}$ for KS $\mathcal{K}$: where $\sigma_k(k) = o$ is the observability value of all $s \in k$, indeed, notice that by construction $\sigma(s) = o$ for all $s \in k$.

Theorem 1. *In the worst case, the KS associated with a POS can have a number of states that is exponential in the number of states of the POS.*

Proof (Sketch). Figure 3 shows an example where the size of the set K is $2^S - 1$ (the empty set is not generated). □

Belief state composition. Let $\mathcal{P}_\mathcal{C} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ be a POS community, we call KS community, the community $\mathcal{K}_\mathcal{C} = \{\mathcal{K}_1, \ldots, \mathcal{K}_n\}$ of KSs, where each $\mathcal{K}_i$ is the KS associated with $\mathcal{P}_i$. We call *belief state trace* for the POS community $\mathcal{P}_\mathcal{C}$ the trace of the associated KS community $\mathcal{K}_\mathcal{C}$. Similarly we define *belief state histories*. We now analyze the relationship between $\mathcal{P}_\mathcal{C}$'s observed history and belief state history. Given a belief state history, we can trivially obtain the observed one by simply applying the σ_k function to all states. Actually, also the vice versa is true: given an observed history, i.e. a sequence of actions and observations, there is only one corresponding belief state history. Indeed, we can define a *bijective function* $g : \mathcal{H}_{obs} \to \mathcal{H}_{ks}$, from

the set of observable histories to the set of belief state histories of $\mathcal{P}_\mathcal{C}$. In other words, let $\mathcal{P}_\mathcal{C}$ be a POS community and $\mathcal{K}_\mathcal{C}$ the respective KS community. *Observing the evolution of $\mathcal{P}_\mathcal{C}$, we can reconstruct the (belief) state in which every service $\mathcal{K}_i$ would be if the same actions are delegated to corresponding services in $\mathcal{K}_\mathcal{C}$.*

Function $g(h_{obs}) = h_{ks}$ can be defined inductively:

- Base case: $h^0_{ks} = \langle k_{1,0}, \ldots, k_{n,0}\rangle = \langle\{s_{1,0}\}, \ldots, \{s_{n,0}\}\rangle$, the KS community initial state;
- Recursive case: let $h^{j+1}_{obs} = h^j_{obs} \xrightarrow{a^{j+1},v^{j+1}} \langle o^{j+1}_1, \ldots, o^{j+1}_n\rangle$, let $last(h^j_{obs}) = \langle o^j_1, \ldots, o^j_n\rangle$, and let $last(h^j_{ks}) = \langle k^j_1, \ldots, k^j_n\rangle$ then: $h^{j+1}_{ks} = h^j_{ks} \xrightarrow{a^{j+1},v^{j+1}} \langle k^{j+1}_1, \ldots, k^{j+1}_n\rangle$ where: $k^{j+1}_{v^{j+1}} = \{s \in img(k^j_{v^{j+1}}, a) \mid \sigma(s) = o^{j+1}_{v^{j+1}}\}$ and $k^{j+1}_i = k^j_i$ for each $i \neq v^{j+1}$.

Notice that the function $g^{-1}(h_{ks}) = h_{obs}$ is simply obtained by applying the function σ_k to each service's state.

POS orchestrator synthesis. Now we can introduce our main result. We show that if we have a composition P_k of a KS community $\mathcal{K}_\mathcal{C}$ for a target service $\mathcal{S}_t$, then P_k *directly corresponds* (through function g) to a POS composition for $\mathcal{S}_t$ wrt POS community $\mathcal{P}_\mathcal{C}$, and that notably also the vice versa holds. Hence, since we have techniques to synthesize compositions in the case of full observability (recall that KS community is fully observable), we can exploit them in order to solve the synthesis in the partially observable case.

Let $\mathcal{P}_\mathcal{C}$ be a POS community, $\mathcal{K}_\mathcal{C}$ the associated KS community, let $\mathcal{H}_{obs}$ and $\mathcal{H}_{ks}$ be the set of $\mathcal{P}_\mathcal{C}$'s observed histories and belief state histories, respectively, and let $\mathcal{S}_t$ be a target service.

Theorem 2. *If P_k is a composition of $\mathcal{K}_\mathcal{C}$ for $\mathcal{S}_t$, then the POS orchestrator P_{obs} defined as: $\forall a \in \mathcal{A}, \forall h_{ks} \in \mathcal{H}_{ks}, P_{obs}(g^{-1}(h_{ks}), a) = P_k(h_{ks}, a)$, is an POS composition for $\mathcal{S}_t$ wrt $\mathcal{P}_\mathcal{C}$.*

Proof (Sketch). We consider the traces τ of $\mathcal{S}_t$ individually, and show that if P_k for $\mathcal{K}_\mathcal{C}$ realizes τ, then $P_{obs}(g^{-1}(\cdot), \cdot)$ for $\mathcal{P}_\mathcal{C}$ realizes τ as well. We show the claim by induction on the length ℓ of the histories $h_{ks} \in \mathcal{H}^\ell_{ks}$, exploiting the mapping given by the function g^{-1}. □

Theorem 3. *If P_{obs} is a POS composition of $\mathcal{P}_\mathcal{C}$ for $\mathcal{S}_t$, then the orchestrator P_k defined as: $\forall a \in \mathcal{A}, \forall h_{obs} \in \mathcal{H}_{obs}, P_k(g(h_{obs}), a) = P_{obs}(h_{obs}, a)$ is a composition for $\mathcal{S}_t$ wrt $\mathcal{K}_\mathcal{C}$.*

Proof (Sketch). We consider again the traces τ of $\mathcal{S}_t$ individually. We show that if P_{obs} for $\mathcal{P}_\mathcal{C}$ realizes τ, then $P_k(g(\cdot), \cdot)$ for $\mathcal{K}_\mathcal{C}$ realizes τ as well. We show the claim by induction on the length ℓ of the histories $h_{obs} \in \mathcal{H}^\ell_{obs}$, this time exploiting the mapping given by the function g. □

We observe that it is quite natural that a composition for the KS community is a POS composition for the associated POS community, indeed the orchestrator for the KS community appears to have less information to work on wrt a POS orchestrator, and hence to be more constrained in its choice of the service to select. The vice versa is more surprising, indeed even if the POS orchestrator is working on the concrete system, its limited ability of observing the system does not give it more information to work on, than what captured by the belief states. Notice that this is not always the case in presence of infinite behavior (cf., Conclusions).

Using the technique above, we can characterize the computational complexity of POS composition synthesis.

Theorem 4. *Checking whether there exists a POS composition (of a POS community) for a target service is EXPTIME-complete. Synthesizing a POS composition is exponential in the number of services in the POS community and in the number of states in each of such services.*

Proof (Sketch). The upper bound is given by the belief state construction described above and the results on the composition of full observable services in (Sardiña, De Giacomo, and Patrizi 2008). The exponential lowerbound in the number of services is a consequence of the EXPTIME-hardness in (Muscholl and Walukiewicz 2008) and the exponential lowerbound in the number of states of each service is a direct consequence of Theorem 1. □

Observe that exponentiality of the belief state construction and that of the basic composition technique do not combine, this is different from the case of conformant planning with partial observability, where the belief state construction does combine with basic planning (Rintanen 2004).

Implementation

Next to tame the exponential cost due to the partial observability, we adopt a synthesis procedure based on symbolic model checking over a game-structure (Patrizi 2009), that delegates the belief state construction for the KS services to the symbolic manipulation of the model checker. We introduce *game structures* (Piterman, Pnueli, and Sa'ar 2006) and show: *(i)* how they can be used for composition under partial observability and *(ii)* how winning strategies for such games can be computed and exploited to get (all) compositions.[1]

Game structures. We specialize *game structures* proposed in (Piterman, Pnueli, and Sa'ar 2006) to deal with synthesis problems for invariant properties. These structures describe games between two adversaries: the *environment* and the *system*. We can control the system but not the environment. Following (Piterman, Pnueli, and Sa'ar 2006), we define a *safety-game structure* (or □-game structure or □-GS, for short) as a tuple $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box\varphi\rangle$, where:

- $\mathcal{V} = \{v_1, \ldots, v_n\}$ is the set of *state variables*, respectively ranging over *finite* domains $V_1, \ldots, V_n$. Wlog, let $\mathcal{X} = \{V_1, \ldots, V_m\}$ and $\mathcal{Y} = \mathcal{V} \setminus \mathcal{X}$. Full valuations, or *game states*, of $\mathcal{X}$'s and $\mathcal{Y}$'s variables are represented as $\langle \vec{x}, \vec{y}\rangle \in V = (V_1 \times \ldots \times V_n)$;
- $\mathcal{X} \subseteq \mathcal{V}$ is the set of *environment* variables. X is the set of all $\mathcal{X}$ variables valuations and each $\vec{x} \in X$ is an *environment state*;
- $\mathcal{Y} = \mathcal{V}/\mathcal{X}$ is the set of *system* variables. Y is the set of all $\mathcal{Y}$ variables valuations and $\vec{y} \in Y$ is a *system state*;
- Θ is a boolean combination of (atomic) expressions $(v_k = \bar{v}_k)$ (with $v_k \in \mathcal{V}$ and $\bar{v}_k \in V_k$) representing game's initial states. Given a *(game) state* $\langle \vec{x}, \vec{y}\rangle \in V$, we write

[1] Throughout the rest of the paper, we assume to deal with infinite-run TSs, possibly obtained by introducing fake loops on final states, as customary in LTL verification/synthesis.

$\langle \vec{x}, \vec{y} \rangle \models \Theta$ iff state $\langle \vec{x}, \vec{y} \rangle$ *satisfies* all Θ assignments. $V_\theta = \{\langle \vec{x}, \vec{y} \rangle \in V \mid \langle \vec{x}, \vec{y} \rangle \models \Theta\}$ is the set of *(game) initial states*;

- $\rho_e \subseteq \mathcal{X} \times \mathcal{Y} \times \mathcal{X}$ is the *environment transition relation* which relates a current game state to a possible next environment state;
- $\rho_s \subseteq \mathcal{X} \times \mathcal{Y} \times \mathcal{X} \times \mathcal{Y}$ is the *system transition relation*, which relates a game state plus a next environment state to a next system state;
- $\Box\varphi$ is a formula representing the invariant property to be guaranteed, where φ has the same form as Θ.

A game state $\langle \vec{x}', \vec{y}' \rangle$ is a *successor* of $\langle \vec{x}, \vec{y} \rangle$, $\langle \vec{x}, \vec{y} \rangle \rightarrow \langle \vec{x}', \vec{y}' \rangle$ for short, if $\rho_e(\vec{x}, \vec{y}, \vec{x}')$ and $\rho_s(\vec{x}, \vec{y}, \vec{x}', \vec{y}')$. A *play* of G is a maximal sequence of states $\eta : \langle \vec{x}_0, \vec{y}_0 \rangle \langle \vec{x}_1, \vec{y}_1 \rangle \cdots$ such that *(i)* $\langle \vec{x}_0, \vec{y}_0 \rangle \models \Theta$ and *(ii)* $\langle \vec{x}_j, \vec{y}_j \rangle \rightarrow \langle \vec{x}_{j+1}, \vec{y}_{j+1} \rangle$ for each $j \geq 0$.

Given a $\Box$-GS G, in a given state $\langle \vec{x}, \vec{y} \rangle$ of a play, the environment chooses an assignment $\vec{x}' \in X$ such that $\rho_e(\vec{x}, \vec{y}, \vec{x}')$ holds and the system chooses assignment $\vec{y}' \in Y$ such that $\rho_s(\vec{x}, \vec{y}, \vec{x}', \vec{y}')$ holds. A play is said to be *winning (for the system)* if it is infinite and satisfies the winning condition $\Box\varphi$, i.e., φ holds in every play state.

A *strategy* for the system is a partial function $f : X^+ \rightarrow Y$ such that for every sequence $\lambda : \vec{x}_0 \cdots \vec{x}_n$ and for every $\vec{y}_n \in Y$, $\vec{x} \in X$, if $\rho_e(\vec{x}_n, \vec{y}_n, \vec{x})$ then $\rho_s(\vec{x}_n, \vec{y}_n, \vec{x}, f(\lambda, \vec{x}))$ holds. A play $\eta : \langle \vec{x}_0, \vec{y}_0 \rangle \langle \vec{x}_1, \vec{y}_1 \rangle \cdots$ is *compliant* with a strategy f if for all $i \geq 0$, $f(\vec{x}_0 \cdots \vec{x}_i) = \vec{y}_i$.

A strategy f is *winning (for the system)* from a given state $\langle \vec{x}, \vec{y} \rangle$ if all plays starting from $\langle \vec{x}, \vec{y} \rangle$ and compliant with f are so. When such a strategy exists, $\langle \vec{x}, \vec{y} \rangle$ is a *winning state* for the system. The *winning set* W is the set of all system's winning states. A $\Box$-GS is *winning (for the system)* if all initial states are in W.

Given a set $Q \subseteq V$ of game states $\langle \vec{x}, \vec{y} \rangle$, we call the set of *$Q$'s controllable predecessors* the following:

$$\pi(Q) \doteq \{\langle \vec{x}, \vec{y} \rangle \in V \mid \forall\, \vec{x}'.\rho_e(\vec{x}, \vec{y}, \vec{x}') \rightarrow \exists\, \vec{y}' \rho_s(\vec{x}, \vec{y}, \vec{x}', \vec{y}') \wedge \langle \vec{x}, \vec{y} \rangle \in Q\}$$

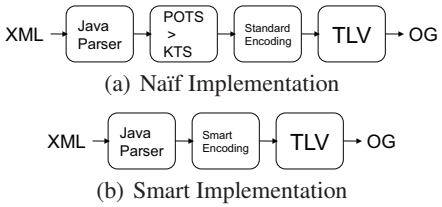
Intuitively, $\pi(Q)$ is the set of states from which the system can force the play to reach a state in Q, no matter how the environment evolves. In other words, for each $\pi(Q)$ state there exists a *one-step* strategy able to reach a state in Q.

Algorithm 1 WIN – Computes system's maximal set of winning states in a $\Box$-GS

```
1: W := {⟨x⃗, y⃗⟩ ∈ V | ⟨x⃗, y⃗⟩ ⊨ φ}
2: repeat
3:     W' := W;
4:     W := W ∩ π(W);
5: until (W' = W)
6: return W
```

Based on such a notion, Algorithm 1 computes the set of all system's winning states of a $\Box$-GS, through a fixpoint computation consisting of iterative applications of π. Theorem 5 shows its correctness.

Theorem 5 (Piterman, Pnueli, and Sa'ar 2006). *Let $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box\varphi \rangle$ be a $\Box$-GS as above and W be obtained as in Algorithm 1. Given a state $\langle \vec{x}, \vec{y} \rangle \in V$, a system's winning strategy f starting from $\langle \vec{x}, \vec{y} \rangle$ exists iff $\langle \vec{x}, \vec{y} \rangle \in W$. Consequently, G is winning for the system iff $V_\Theta \subseteq W$.*



(a) Naïf Implementation

(b) Smart Implementation

Figure 4: Implementations

In practice, from W, one can inductively define a system's winning strategy f by requiring that *(i)* $f(\vec{x}_0) = \vec{y}_0$ for each $\langle \vec{x}_0, \vec{y}_0 \rangle \in V_\Theta$ and *(ii)* for each play $\eta : \langle \vec{x}_0, \vec{y}_0 \rangle \cdots \langle \vec{x}_n, \vec{y}_n \rangle$ compliant with f and environment state $\vec{x}_{n+1} \in X$ such that $\rho_e(\vec{x}_n, f(\vec{x}_0, \ldots, \vec{x}_n), \vec{x}_{n+1})$, $\langle \vec{x}_{n+1}, f(\vec{x}_0, \ldots, \vec{x}_n, \vec{x}_{n+1}) \rangle \in W$.

This approach to synthesis based on model checking game structures is implemented in TLV (Piterman, Pnueli, and Sa'ar 2006): a software for verification and synthesis of LTL specifications, based on symbolic manipulation of states, which uses Binary Decision Diagrams (BDDs). In particular TLV is able to take as input specification that encodes the environment and the system in SMV and the invariance property in LTL and derives the system's maximal winning set for the corresponding $\Box$-GS.

POS composition synthesis using safety games. Next, we show an encoding a composition problem instance into a game structure so that for each system's winning strategies there exists a composition and viceversa. The procedure is inspired by the one proposed in (Patrizi 2009), based on a reduction of the composition problem instance to a game structure and a subsequent winning set computation, performed by the synthesis system TLV. However, in (Patrizi 2009), the composition problem was faced under full observability assumption. Here, we propose a generalization able to deal with partial observability.

Before giving details, observe that, with a sound and complete procedure available under full observability, one can immediately devise a naïf implementation for POS composition. depicted in Figure 4(a) where, first, each POS, e.g., represented as an XML file, is transformed, by, e.g., an ad-hoc Java program, into its respective KS and, hence, the procedure of (Patrizi 2009) is executed, i.e., the resulting (*fully observable*) composition problem is first encoded into a game structure and then the winning set is computed by TLV. While clearly correct, such an implementation is computationally challenging since the generated KSs can have an exponential size wrt their respective POS's and their manipulation is done explicitly by the Java program.

Our actual implementation is shown in Figure 4(b). The main point is that we delegate the construction of the KSs to TLV, taking advantage of its built in symbolic, OBDD-based, manipulation of transition systems. For each POS, we provide TLV a *compact* description of the respective KS, which is *polynomial* wrt to original POS' size.

Conceptually, our goal is to refine an automaton, the or-

chestrator, capable of selecting, at each step, one among all the available services, so to make it a composition. In other words the *orchestrator* is the object of the synthesis which, in a game structure, plays as the *system*. On the other side the combination of the *target* and the *available services* play as the *environment*.

Let $\mathcal{P}_\mathcal{C} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ be a community and $\mathcal{S}_t$ a target service, where, $\mathcal{P}_i = \langle \mathcal{A}, \mathcal{O}, S_i, s_{i,0}, S_i^f, \varrho_i, \sigma_i \rangle$ ($i = 1 \ldots, n$) and $\mathcal{S}_t = \langle \mathcal{A}, S_t, s_{t,0}, S_t^f, \varrho_t \rangle$. We derive a $\Box$-GS $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_c, \Box\varphi \rangle$, as follows:

- $\mathcal{V} = \{s_t, \vec{k}_1, \ldots, \vec{k}_n, a, ind\}$, where:
  - s_t ranges over $S_t \cup \{init\}$;
  - each $\vec{k}_i$ is a variable array of $|S_i|$ components taking values from $\{0, 1\}$ whose valuations stand for characteristic functions which represent, in turn, elements from 2^{S_i}. The empty set (represented by $\vec{0} = \langle 0, \ldots, 0 \rangle$) stands for a special initial value introduced for convenience;
  - a ranges over $\mathcal{A} \cup \{init\}$;
  - ind ranges over $\{1, \ldots, n\} \cup \{init\}$;

  with the following intuitive meaning: each complete valuation of $\mathcal{V}$ represents *(i)* current target's (s_t) and community services' (variables $\vec{k}_1, \ldots, \vec{k}_n$) states, *(ii)* the operation to be performed next (a) and *(iii)* the available service selected to perform it (ind). Special value $init$ and array $\vec{0}$ are used so as to have a fixed initial state;
- $\mathcal{X} = \{s_t, \vec{k}_1, \ldots, \vec{k}_n, a\}$ is the set of environment variables;
- $\mathcal{Y} = \{ind\}$ is the (singleton) set of system variables;
- $\Theta(s_t, \vec{k}_1, \ldots, \vec{k}_n, a, ind) = ((s_t = init) \wedge (\bigwedge_{i=0,\ldots,n}(\vec{k}_i = \vec{0}))) \wedge (a = init) \wedge (ind = init)$;
- in order to define $\rho_e(\mathcal{X}, \mathcal{Y}, \mathcal{X}')$, first let us introduce the following abbreviations

  (i) $op\_possible_i(\vec{k}_i, a) \doteq \bigwedge_{s \in \vec{k}_i} (\bigvee_{s' \in S_i} (s \xrightarrow{a} s' \; in \; \mathcal{P}_i))$

  which denotes whether i-th service can perform requested action a, and

  (ii) $\overrightarrow{imm\_oss_{i,j}}(\vec{k}_i, a) \doteq \begin{cases} \emptyset, \; if \; \neg op\_possible_i(\vec{k}_i, a) \\ \overrightarrow{\Sigma_{i,j}}(\vec{k}_i, a), \; otherwise \end{cases}$

  where $\overrightarrow{\Sigma_{i,j}}(\vec{k}_i, a) \doteq \{s' \in S_i \mid \sigma(s') = j \wedge \exists s \in \vec{k}_i . s \xrightarrow{a} s' \; in \; \mathcal{P}_i\}$, is an array of the same form as all $\vec{k}_i$'s, representing the set of all i-th service states, with same observation j, reachable from belief state $\vec{k}_i$, by performing operation a.

  Then, $\langle \langle init, \vec{0}, \ldots, \vec{0}, init \rangle, init, \langle s_t, \vec{k}_1, \ldots, \vec{k}_n, a \rangle \rangle \in \rho_e$ iff $\vec{k}_i = k_{i,0}$, for $i = 1, \ldots, n$, $s_t = s_{t,0}$;

  if $\vec{k}_i \neq \vec{0}$, with $i = 1, \ldots, n$, $s_t \neq init$, $a \neq init$ and $ind \neq init$ then $\langle \langle s_t, \vec{k}_1, \ldots, \vec{k}_n, a \rangle, ind, \langle s_t', \vec{k'}_1, \ldots, \vec{k'}_n, a' \rangle \rangle \in \rho_e$ iff the followings hold in conjunction:
  1. there exists a transition $s_t \xrightarrow{a} s_t'$ in $\mathcal{S}_t$;
  2. either $\bigvee_{j=0}^{|\mathcal{O}|} \vec{k'}_{ind} = \overrightarrow{imm\_oss_{ind,j}}(\vec{k}_{ind}, a))$ or $\vec{k'}_{ind} = \vec{k}_{ind}$ (service wrongly makes no move, and the error violates the safety condition φ, see below);
  3. $\vec{k}_i = \vec{k'}_i$, for all $i = 1, \ldots, n$ such that $i \neq ind$;
  4. there exists a transition $s_t' \xrightarrow{a'} s_t''$ in $\mathcal{S}_t$ for some s_t'';
- $\langle \langle s_t, \vec{k}_1, \ldots, \vec{k}_n, a \rangle, ind, \langle s_t', \vec{k'}_1, \ldots, \vec{k'}_n, a' \rangle, ind' \rangle \in \rho_s$ iff $ind' \in \{1, \ldots, n\}$;
- Formula $\Box\varphi$ where φ is:

  $\varphi(s_t, \vec{k}_1, \ldots, \vec{k}_n, a, ind) \doteq$
  $\Theta(s_t, \vec{k}_1, \ldots, \vec{k}_n, a, ind) \vee (\bigwedge_{i=1}^n \neg fail_i(\vec{k}_i, a, ind)) \wedge (final_t(s_t) \rightarrow \bigwedge_{i=1}^n final_i(\vec{k}_i))$, with:
  - $fail_i(\vec{k}_i, a, ind) \doteq (ind = i) \wedge \neg op\_possible_i(\vec{k}_i, a)$, encodes the fact that service i has been selected but, in its current state, no transition can take place which executes the requested operation;
  - $final_t(s_t) \doteq \bigvee_{s \in S_t^f} s_t = s$, encodes the fact that the target service is currently in one of its final states;
  - $final_i(\vec{k}_i) \doteq \bigwedge_{s \in \vec{k}_i} s \in S_i^f$, encodes the fact that service $i = 1, \ldots, n$ is currently in one of its final states.

Notably *all formulas above require only the current KS community state, target state, operation and service selection, and can all be computed on-the-fly*. This is exactly what our encoding leads TLV to do.

Next, we show the correctness of the above construction.

Theorem 6. *Let $\mathcal{P}_\mathcal{C} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ be a POS community and $\mathcal{S}_t$ a target service, where $\mathcal{P}_i = \langle \mathcal{A}, \mathcal{O}_i, S_i, s_{i,0}, S_i^f, \varrho_i, \sigma_i \rangle$ ($i = 1 \ldots, n$) and $\mathcal{S}_t = \langle \mathcal{A}, S_t, s_{t,0}, S_t^f, \varrho_t \rangle$. From $\mathcal{P}_\mathcal{C}$ and $\mathcal{S}_t$ derive: a $\Box$-GS $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box\varphi \rangle$ as shown above. Let $W \subseteq V$ be the maximal set of system's winning states for G. Then $\langle init, \vec{0}, \ldots, \vec{0}, init, init \rangle \in W$ if and only if there exists an POS orchestrator for $\mathcal{P}_\mathcal{C}$ that is an POS composition of $\mathcal{S}_t$.*

Proof (Sketch). Consequence of Theorems 2 and 3, of above encoding ability to actually generate the KS community, and correctness of the approach for fully observable services (cf. (Sardiña, De Giacomo, and Patrizi 2008; Patrizi 2009)). $\Box$

Interestingly, W at hand allows for computing all POS compositions in one shot. This is tightly related to the notion of *orchestrator generator*, and shows that one can construct so called "just-in-time" compositions even in the case of partial observability, cf. (Sardiña, De Giacomo, and Patrizi 2008).

Theorem 7. *For $\mathcal{P}_\mathcal{C}$, $\mathcal{S}_t$ and G as above, let W be the system's winning set for G with $\langle init, \vec{0}, \ldots, \vec{0}, init, init \rangle \in W$, and let*

$$\omega(\langle s_t, \vec{k}_1, \ldots, \vec{k}_n \rangle, a) = \{i \in \{1, \ldots, n\} \mid \langle s_t, \vec{k}_1, \ldots, \vec{k}_n, a, i \rangle \in W\}$$

Then a POS orchestrator P_{obs} is an POS composition iff it is built from W as follows: $\forall a \in \mathcal{A}$ and $\forall h_{st} \in \mathcal{H}$ we

```
MODULE Main
VAR
  env: system Env(sys.index);
  sys: system Sys;
DEFINE
  good := (sys.initial & env.initial)|!(env.failure);
```

Figure 5: TLV main module

have:

$$P_{obs}(obs(h_{st}), a) = \begin{cases} i \in \omega(last(g(obs(h_{st}), a) \neq \emptyset \\ u \quad otherwise \end{cases}$$

Proof (Sketch). Consequence of Theorem 6, and of the fact that the maximal winning set maintains all possible indexes that lead to a composition. See (Sardiña, De Giacomo, and Patrizi 2008; Patrizi 2009) for more details. □

Figure 5 shows the basic blocks of a sample encoding for a composition problem with 3 available services. Module `Main` wraps up all other modules and represents the whole game. It consists of two submodules (here declared as `system`), `sys` and `env`, which encode, respectively, the environment and the system in the game structure. Goal formula `good` (i.e., the invariant property) is a combination of subformulae `initial` and `failure` of modules `sys` and `env`, directly obtained from the goal formula in the □-GS representation. Observe that `env` and `sys` evolve synchronously, the former choosing the operation and the latter selecting the service for its execution. The transition relation in module `Sys` encodes an *unconstrained controller*, able to output, at each step, any available service index in the interval $[1, n]$. The synthesis' objective is to restrict such a relation so to obtain a winning strategy.

Conclusion

In this paper, we devised techniques and results for composing services that are partially controllable and partially observable, which corresponds, using Planning terminology, to the conformant form of the composition proposed in (De Giacomo and Sardiña 2007; Sardiña, De Giacomo, and Patrizi 2008). Our procedure works with belief states, which is a standard approach when dealing with conformant planning, and has already been used in the context of behavior-based service composition (Pistore, Traverso, and Bertoli 2005; Pistore et al. 2005). Observe we often give for discounted that such a construction is sound and complete, but in general it is only sound (Sardiña et al. 2006). It becomes complete when plans do not need include loop. Here we do include loops, so completeness is not a priori granted. We do get completeness here, without contradicting the results in (Sardiña et al. 2006), because our "plans" are not based on reachability, but on a sort of maintenance property (maintaining the simulation). We close the work by mentioning that it would be interesting to compare our approach to partial observability, based on belief states, to the one in (Balbiani, Cheikh, and Feuillade 2008), inherited by discrete event control (Arnold, Vincent, and Walukiewicz 2003), where partial observability comes in the form of actions from the available services that cannot be observed by the orchestrator. This would be of interest also in other (conformant) planning settings.

References

Alur, R.; Henzinger, T. A.; and Kupferman, O. 2002. Alternating-time temporal logic. *JACM* 49(5):672–713.

Arnold, A.; Vincent, A.; and Walukiewicz, I. 2003. Games for synthesis of controllers with partial observation. *Th. Comp. Sc.* 1(303):7–34.

Balbiani, P.; Cheikh, F.; and Feuillade, G. 2008. Composition of interactive web services based on controller synthesis. In *Proc. of IEEE SERVICES'08*, 521–528.

Berardi, D.; Calvanese, D.; De Giacomo, G.; Lenzerini, M.; and Mecella, M. 2005. Automatic service composition based on behavioural descriptions. *IJCIS* 14(4):333–376.

De Giacomo, G., and Sardiña, S. 2007. Automatic Synthesis of New Behaviors from a Library of Available Behaviors. In *Proc. of IJCAI 2007*, 1866–1871.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kauffman.

Goldman, R. P., and Boddy, M. S. 1996. Expressive planning and explicit knowledge. In *Proc. of AIPS'96*, 110–117.

Hatzi, O.; Meditskos, G.; Vrakas, D.; Bassiliades, N.; Anagnostopoulos, D.; and Vlahavas, I. P. 2008. A synergy of planning and ontology concept ranking for semantic web service composition. In *Proc. of IBERAMIA'08*, 42–51.

Hull, R. 2005. Web Services Composition: A Story of Models, Automata, and Logics. In *Proc. of SCC'05*.

Klusch, M., and Gerber, A. 2006. Fast composition planning of owl-s services and application. In *Proc. of ECOWS'06*, 181–190.

Lustig, Y., and Vardi, M. Y. 2009. Synthesis from Component Libraries. In *Proc. of FOSSACS'09*.

McIlraith, S. A., and Son, T. C. 2002. Adapting Golog for Composition of Semantic Web Services. In *In Proc. of KR'02*, 482–496.

Muscholl, A., and Walukiewicz, I. 2008. A Lower Bound On Web Services Composition. *LMCS* 4(2).

Patrizi, F. 2009. *Simulation-Based Techniques for Automated Service Composition*. Ph.D. Dissertation, SAPIENZA Univ. Roma.

Pistore, M.; Marconi, A.; Bertoli, P.; and Traverso, P. 2005. Automated Composition of Web Services by Planning at the Knowledge Level. In *Proc. of IJCAI 2005*.

Pistore, M.; Traverso, P.; and Bertoli, P. 2005. Automated Composition of Web Services by Planning in Asynchronous Domains. In *Proc. of ICAPS 2005*, 2–11.

Piterman, N.; Pnueli, A.; and Sa'ar, Y. 2006. Synthesis of Reactive(1) Designs. In *VMCAI'06*, 364–380.

Rintanen, J. 2004. Complexity of Planning with Partial Observability. In *Proc. of ICAPS'04*, 345–354.

Sardiña, S.; De Giacomo, G.; Lespérance, Y.; and Levesque, H. J. 2006. On the Limits of Planning over Belief States under Strict Uncertainty. In *Proc. of KR '06*, 463–471.

Sardiña, S.; De Giacomo, G.; and Patrizi, F. 2008. Behavior Composition in the Presence of Failure. In *Proc. of KR'08*.

Stroeder, T., and Pagnucco, M. 2009. Realising Deterministic Behaviour from Multiple Non-Deterministic Behaviours. In *Proc. of IJCAI'09*.

van der Hoek, W.; Roberts, M.; and Wooldridge, M. 2007. Social laws in alternating time: effectiveness, feasibility, and synthesis. *Synthese* 156(1):1–19.

Vardi, M. Y. 2008. From Church and Prior to PSL. In *25 Years of Model Checking*, 150–171.